INSTITUTO FEDERAL DE CIÊNCIA E TECNOLOGIA DE PERNAMBUCO

Campus Recife

Departamento Acadêmico de Cursos Superiores.

Tecnologia em Análise e Desenvolvimento de Sistemas

DAVID RAMOS DA SILVA

HEBER LEANDRO DA LUZ SILVA

AVALIAÇÃO DO DESEMPENHO DO WEBASSEMBLY: UM ESTUDO DE CASO UTILIZANDO RECONHECIMENTO DE DIGITOS MANUSCRITOS.

Recife

2023

DAVID RAMOS DA SILVA

HEBER LEANDRO DA LUZ SILVA

AVALIAÇÃO DO DESEMPENHO DO WEBASSEMBLY: UM ESTUDO DE CASO UTILIZANDO RECONHECIMENTO DE DIGITOS MANUSCRITOS.

Projeto de Trabalho de conclusão de curso em Análise e Desenvolvimento de Sistema do Instituto Federal de Ciência e Tecnologia de Pernambuco, como requisito para obtenção do título Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientadora: Profa. Ms(a) Renata Freire de Paiva Neves

Recife

2023

Ficha catalográfica elaborada pela bibliotecária Danielle Castro da Silva CRB4/1457

S586a

2023 Silva, David Ramos da

Avaliação do desempenho do webassembly: um estudo de caso utilizando reconhecimento de dígitos manuscritos. / David Ramos da Silva; Heber Leandro da Luz Silva. --- Recife: Os autores, 2023.

54f. il. Color.

Trabalho de Conclusão (Curso Superior Tecnológico em Análise e Desenvolvimento de Sistemas) – Instituto Federal de Pernambuco, Recife, 2023.

Inclui Referências, Glossário e Apêndices.

Orientadora: Profa. Ms(a). Renata Freire de Paiva Neves.

1. Aplicações web. 2. Webassembly.. 3. Javascript . 4. Emscripten. I. Título. II. Neves, Renata Freire de Paiva (orientadora). III. Instituto Federal de Pernambuco.

CDD 005.1

Trabalho de Conclusão de Curso apresentado pelos estudantes **David Ramos da Silva e Heber Leandro da Luz Silva** à coordenação de Análise e Desenvolvimento de Sistemas, do Instituto Federal de Pernambuco, sob o título de "**AVALIAÇÃO DO DESEMPENHO DO WEBASSEMBLY: UM ESTUDO DECASO UTILIZANDO RECONHECIMENTO DE DIGITOS MANUSCRITOS"**, orientados pela **Profa. Ms(a). Renata Freire de Paiva Neves** e aprovados pela banca examinadora formada pelos professores:

Recife, 27 de abril de 2023.

_____

Profa. Ms(a). Renata Freire de Paiva Neves

CTADS/DACS/IFPE

_____

Prof. Ms. Hilson Gomes Vilar de Andrade

DACT/IFPE

_____

Prof. Dr. Eduardo de Melo Vasconcelos

CTADS/DACS/IFPE

Recife

2022

Dedicamos este trabalho a Deus.

O que primeiro nos predestinou.

AGRADECIMENTOS

Agradecemos a Deus que nos predestinou para este momento.

Também somos gratos à instituição de ensino e ao seu corpo docente por todo o apoio e parceria que nos permitiu chegar até aqui.

Queremos agradecer especialmente a Profa. Ms Renata que mesmo após todo o tempo de desenvolvimento e espera, não desistiu de nós e nos estimulou a realizar uma abordagem técnica e precisa. Admiramos todo o conhecimento vasto e multidisciplinar que você tem, Renata.

Aos nossos pais e familiares pelo amor, apoio e incentivo.

Agradecemos aos nossos amigos Gean e Giovanni que estiveram conosco desde a concepção da ideia.

Eu, David, agradeço a minha namorada Maria, que me acompanhou nas noites difíceis. Ao meu irmão, Oséias que pacientemente me ouviu explicar os conceitos deste trabalho um milhão de vezes. Agradeço também às minhas queridas amigas Luana e Nicoly que estão apoiando minhas ideias há muito tempo.

Eu, Heber, agradeço a minha família, em especial a minha mãe, que sempre me incentivou e me direcionou a escolha desse curso que hoje é minha profissão.

"Low-level programming is good for the programmer's soul"

John Carmack

RESUMO

O presente trabalho tem como objetivo apresentar um estudo sobre o desempenho em questão de velocidade de aplicações web que utilizam webassembly. Webassembly é uma linguagem e um formato binário de arquivo que promete ganhos de velocidade significativos em relação ao javascript. No estudo, foram desenvolvidas duas aplicações capazes de processar e classificar imagens com dígitos manuscritos. No primeiro software, todos os algoritmos de pré-processamento de imagem e a rede neural para a classificação foram implementados na linguagem C++ e compilados para webassembly utilizando o Emscripten; na segunda aplicação, os mesmos algoritmos foram escritos da maneira tradicional utilizando apenas Javascript. Ao término do estudo, testes de benchmarking foram realizados utilizando um dataset sintético no estilo MNIST. Em conclusão, o estudo sobre WebAssembly mostrou que ele tem resultados melhores nos cenários apresentados na maioria dos navegadores testados, permitindo aproveitar o poder computacional dos usuários do *client-side* e reduzindo a sobrecarga de chamadas ao *backend*. Os testes de *benchmarking* foram realizados nos browsers Google Chrome, Microsoft Edge, Mozilla Firefox e Brave. Em resumo, o WebAssembly é uma tecnologia promissora e que merece ser explorada por desenvolvedores e empresas, para aproveitar todo seu potencial e levar o desempenho de aplicações web ao próximo nível.

Palavras-chave: webassembly; javascript; emscripten; visão computacional; inteligência artificial.

ABSTRACT

The goal of the current work is to present a study on the effectiveness of web-based applications that use webassembly. A language and binary file format called Webassembly promises significant speed gains over javascript. In the study, two applications capable of processing and classifying images with handwritten data were developed. All of the image preprocessing algorithms and neural network algorithms for classification were written in C++ for the first application and compiled for webassembly using Emscripten; the same algorithms were written in Javascript for the second application. At the conclusion of the study, benchmarking tests were conducted using a fictitious dataset in the MNIST style. In conclusion, the study on WebAssembly showed that it has superior performance results in most browsers, allowing us to take advantage of the computational power of client-side users and reducing the overload of calls to the backend. The benchmarking tests were carried out on the browsers Google Chrome, Microsoft Edge, Mozilla Firefox and Brave. In short, WebAssembly is a promising technology and deserves to be explored by developers and companies, to take advantage of all its potential and take the performance of web applications to the next level.

Keywords: webassembly; javascript; emscripten; computer vision; artificial intelligence.

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

WASM	Webassembly
WAT	Webassembly Textual Format
JS	Javascript
JIT	Just-in-time
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
ROI	Region of interest
WSL 2	Windows Subsystem for Linux 2

SUMÁRIO

1 INTRODUÇÃO

Há 25 anos surgia a linguagem que mais tarde se tornaria a principal linguagem para o desenvolvimento de aplicações web: o Javascript[1] (JS). Criado pela Netscape a linguagem tinha um potencial de crescimento enorme, contudo, com o domínio completo no chamado *client-side*, o javascript trouxe consigo também suas limitações. Por ser uma linguagem interpretada, o JS não era performático o suficiente para o desenvolvimento de aplicações web complexas (ANDREASEN et al, 2017).

Em 2017, o mundo da tecnologia presenciou o início da resolução desse problema com o lançamento do webassembly[2] (WASM). A proposta é simples: tornar códigos escritos em outras linguagens compatíveis como C/C++[3] ou Rust[4], completamente portáveis para a web, rápidos e garantindo a solidez do modelo de *sandbox* do *browser* que permite que o usuário navegue por qualquer site em segurança.

A grande vantagem do uso do webassembly é se aproveitar da velocidade de execução que as linguagens de baixo nível oferecem, possibilitando a criação de aplicações web que utilizam módulos, funções, classes e utilitários escritas em linguagem nativa (MOZILLA, 2017?c). Como mostrado ao longo deste trabalho, o desenvolvimento de algoritmos de processamento de imagem e o uso de técnicas de visão computacional são conhecidos por exigirem muito poder computacional das máquinas, e os algoritmos escritos precisam ser eficientes. Em resumo, os ganhos em desempenho e inovação que a tecnologia trouxe foram fatores importantes para a escolha do webassembly em conjunto com o processamento de imagens e a visão computacional, para identificar a real capacidade da ferramenta e como ela pode trazer um ganho de desempenho para aplicações web, utilizando os recursos do client-side.

---

[1] JavaScript, Mozilla Corp., Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[2] WebAssembly, Disponível em: https://webassembly.org/

[3] C++, C++ Foundation, Disponível em: https://isocpp.org/

[4] Rust, Disponível em: https://www.rust-lang.org/

1.1 Objetivo Geral

Verificar a desempenho em termos de velocidade de execução do webassembly no *client-side* na tarefa custosa do processamento de imagens para a classificação de dígitos manuscritos.

1.2 Objetivos Específicos

- Analisar a viabilidade do desenvolvimento de aplicações utilizando o webassembly.
- Desenvolver duas aplicações web equivalentes para o processamento e classificação de dígitos manuscritos. A primeira usando webassembly e a segunda usando javascript
- Implementar os algoritmos utilizados nas etapas de pré-processamento, extração de características e segmentação das imagens.
- Analisar os resultados dos testes de benchmarking de cada uma das aplicações a fim de verificar a viabilidade do desenvolvimento web mais performático com o webassembly.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta um breve referencial teórico sobre as áreas envolvidas neste projeto, destacando os desafios do uso do webassembly como uma possível solução performática para as tarefas que envolvem o processamento de imagens no *front-end* para serem aplicadas a uma rede neural artificial classificadora de dígitos manuscritos.

2.1 Modelo cliente-servidor

Para entender o propósito do webassembly nesta pesquisa, precisamos saber como funciona o modelo cliente-servidor. As aplicações web atuais tem o seu *client-side* composto por alguns componentes que são responsáveis por renderizar, dar estilos às páginas, além de entregar funcionalidades úteis; Componentes esses que podem ser gerados pelo código do *server-side* (OCARIZA et al, 2013)**.**

O *server-side* ou servidor é parte da aplicação que não é visível pelo usuário final, ou seja, toda lógica de regra de negócios fica sob sua responsabilidade. Nesse modelo o cliente é o solicitante e o servidor é aquele que provê o serviço requisitado. Deste modo, é comum o servidor ser responsável pelo processamento de dados e enviar resultados de volta ao cliente (OLUWATOSIN ,2014). Esse modelo também é atualmente chamado de WEB 2.0, e é o modo como as pessoas conhecem o desenvolvimento web hoje.

Com o passar dos anos e com o avanço da tecnologia, é notável o aumento na capacidade de processamento dos computadores; trazendo para o desenvolvimento web no modelo apresentado, os servidores ganharam melhorias e com isso a possibilidade de execução de tarefas mais complexas, isso é um fator muito forte para a manutenção do modelo cliente-servidor. Contudo, outro fator frequentemente esquecido é que os computadores pessoais do *client-side*, também evoluíram e, muitas vezes, o poder de processamento que pode ser aproveitado na máquina dos usuários, é delegado ao servidor. Isso não é necessariamente um problema, mas pode ser um desperdício de recursos.

Uma aplicação que usa o modelo cliente-servidor tradicional, envia na primeira requisição, os componentes de HTML, CSS e Javascript, esses por sua vez, precisam ser baixados e "montados" no *browser* do usuário que está navegando na página. Esse download por si só já é responsável por grande parte do consumo de banda, principalmente se o site fizer muito uso de javascript (OSMANI, 2017), contudo essa situação pode piorar se a aplicação fizer diversas requisições ao servidor para delegar tarefas como, por exemplo, a execução de algum algoritmo de inteligência artificial, mudança de filtro de uma imagem, processamento de texto ou transformações em frames de vídeo.

Como a proposta do WASM é ser um formato binário de baixo nível que é executado no *client-side*, podemos aproveitar de alguns fatores para tentar melhorar o desempenho de aplicações que necessitam de muito processamento que antes poderia ser responsabilidade do servidor.

Sendo assim, a proposta é de aproveitar do poder de processamento do computador do usuário no *client-side*, utilizando o webassembly, para o desenvolvimento de aplicações web mais rápidas e mais econômicas no quesito consumo de banda. Essa melhoria no *client-side* diminuiria a carga de requisições que o servidor recebe e este ficaria responsável apenas por tarefas que estão relacionadas a regras de negócio ou segurança, por exemplo.

2.1.1 Da criação dos navegadores ao nascimento do Javascript

Esclarecendo os termos, o cliente ou *client-side* é normalmente representado por navegadores web como o Google Chrome, Opera, Safari e Firefox, sendo assim, todas às vezes que acessamos um site, o navegador envia uma requisição ao servidor ou *server-side* e este, por sua vez, responde com o conteúdo solicitado, no caso, as instruções para "desenhar" o site na tela do usuário. A história de criação dos navegadores é cheia de intrigas e acusações de monopolismo (BODILY, 2009), mas focaremos na explicação da quase onipresença do javascript por toda web.

Na consolidação do mercado de navegadores, grandes players tentaram garantir sua fatia de mercado, mas a empresa Netscape foi a mais bem sucedida nessa primeira geração. A empresa foi a responsável pela criação do javascript, a primeira linguagem de *scripting* dos *browsers*. Pela popularidade do produto, várias outras empresas se adaptaram ao padrão de suporte à linguagem e com o passar do

tempo e com ajuda da padronização do javascript pela ECMA International, a linguagem se tornou a mais popular das aplicações Web (MOZILLA, 2016?). Contudo, apesar de ser uma linguagem simples de ser compreendida, visto os requerimentos cada vez mais complexos dos websites criados, o JS possui limitações.

Nos anos 2000, as aplicações web não poderiam ser muito complexas, pois não existiam técnicas sofisticadas para que isso fosse possível. Em vez disso, as aplicações web deveriam ser desenvolvidas usando softwares de design de *layouts* como o Photoshop[5], Corel Draw[6] ou DreamWeaver[7]. Tais ferramentas eram difíceis de manipular, caras e muitas vezes não geravam o resultado idêntico a visualização quando o site era disponibilizado na Internet (JANETAKIS, 2018).

Esses problemas mais remotos da história do desenvolvimento web foram resolvidos com os avanços na padronização das tecnologias usadas no *client-side* e no avanço geral no entendimento da engenharia de software. Com isso, o mundo conseguiu presenciar a construção de sites e aplicações na Internet verdadeiramente complexas que poderiam resolver problemas importantes do mundo moderno. Redes sociais, processadores de texto e editores de vídeos já eram projetos possíveis e sólidos.

Contudo, o desenvolvimento de aplicações web modernas, apesar de utilizar um javascript já amadurecido e otimizado, ainda possui gargalos (ANDREASEN, 2017). Sistemas muito complexos podem se ver limitados principalmente pelo desempenho de execução da linguagem.

Os navegadores web modernos executam o javascript dentro de uma máquina virtual para garantir a segurança ao usuário ao navegar pela web. Essa máquina virtual possui um recurso muito importante chamado Javascript *engine*, ou interpretador, componente essencial para a interpretação e execução do código JS. Dessa forma, somos apresentados a dois novos conceitos: linguagens interpretadas e linguagens compiladas. O javascript faz parte do primeiro grupo e para posteriormente entendermos a necessidade do webassembly é importante compreender primeiro os dois processos. Os processos são sobretudo, duas maneiras

---

[5] Adobe Inc. Photoshop. 2022. Disponível em: https://www.adobe.com/products/photoshop.html

[6] Corel Corporation. Corel Draw. 2022. Disponível em: https://www.coreldraw.com/br

[7] Adobe Inc. DreamWeaver. 2023. Disponível em: https://www.adobe.com/products/dreamweaver

específicas de traduzir um código escrito por um humano para uma linguagem de baixo nível que somente o computador consegue entender

2.1.2 Diferença entre compilação e interpretação

O interpretador é um software responsável por interpretar instrução por instrução durante a execução do código que lhe é passado, ou seja, o código que é escrito em uma linguagem de alto nível como javascript, é executado instantaneamente após a tradução. Sendo assim, toda vez que determinado trecho for chamado, acontecerá a interpretação e execução novamente (KWAME, MARTEY e CHRIS, 2017).

O compilador também é um software capaz de fazer a tradução de uma linguagem de programação para a linguagem de máquina, contudo, diferentemente do interpretador, o compilador traduz o código inteiro de uma só vez para seu equivalente em linguagem de máquina específica para a arquitetura do computador onde o programa será executado. Compilar o código inteiro de uma só vez é um pouco mais lento e além disso, é um pré-requisito mais espaço em disco e em memória, por que durante a compilação, o compilador gera arquivos binários (ou arquivos objetos) intermediários que precisam ser ligados na etapa de linkedição (KWAME, MARTEY e CHRIS, 2017).

Apesar disso, quando o executável é gerado, ele consegue ter um desempenho superior, em diversos casos, que os programas interpretados. Esse ganho de velocidade pode ser explicado pelas diversas otimizações que o compilador consegue fazer no código de máquina gerado.

2.1.3 *Just-in-time compiler*

As empresas que desenvolviam as *engines* de interpretação do javascript, que por sua vez, são utilizadas nos browsers modernos, adicionaram uma nova e interessante etapa nos interpretadores. Essa melhoria nos motores de interpretação do JS já era conhecida há muito tempo pelo nome de *just-in-time compiler*, ou JIT.

Os compiladores JIT fazem uso de técnicas que unem o melhor dos dois mundos da interpretação e compilação clássicas, eles misturam conceitos a fim de melhorar o desempenho dos códigos interpretados. (CLARK, 2017).

Utilizando como exemplo a implementação da *engine* mais famosa do mercado, chamada V8 que é embutida no Google Chrome, essa tecnologia apresentou um novo módulo chamado de monitor ou *profiler* que é responsável por monitorar a quantidade de vezes que um determinado fragmento de código é executado.

Essa contagem é extremamente importante, pois na prática, conseguimos resolver um dos maiores problemas na interpretação: a repetição da tradução de código que já foi executada anteriormente.

Em linhas gerais, o monitoramento do código que está sendo executado, permite que as *engines* criem marcações nos trechos de código que estão sendo repetidamente executados e a quantidade de vezes em que essa execução aconteceu nos dará indicadores que nos falam sobre a “temperatura” do trecho do código. Apesar da ideia ser implementada de uma maneira diferente em cada um dos *browsers*, o conceito geral continua sendo o mesmo: Caso o código tenha sido executado diversas vezes, o interpretador o marcará como “quente”.

De maneira resumida, Clark explica que o raciocínio é o seguinte: todo trecho de código marcado como “quente” será passado ao JIT e compilado, evitando dessa forma, o processo de repetição de interpretação. Além disso, códigos marcados como “muito quentes” passam pela compilação e ainda recebem diversas otimizações. Por fim, a *engine* de execução armazenará o trecho de código que foi compilado e o usará quando a execução do mesmo for requisitada.

Dessa maneira, os compiladores *just-in-time* conseguem unir as vantagens de velocidade e otimização do processo de compilação em linguagens interpretadas. O ganho final na execução do javascript é claro, contudo, todo o processo de monitoramento e compilação dos trechos mais executados precisa de mais tempo para finalizar, sendo assim, é necessário investir o valioso tempo de execução para ganhar em quesitos de velocidade. Essa troca, pode gerar gargalos nas aplicações web.

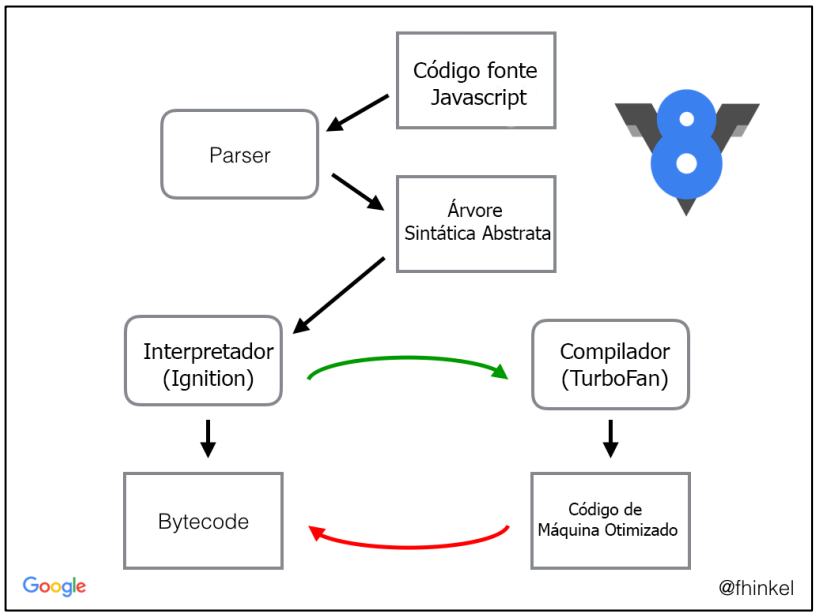
Explicadas as particularidades dos processos de compilação e interpretação, conseguimos entender a maneira que os *browsers* lidam com os códigos escritos pelos desenvolvedores e como acontece a tradução e a execução dessas instruções.

Apesar das melhorias implementadas no processo de interpretação e execução do javascript, uma *engine* não funciona somente com este módulo.

Para entendermos melhor, existem diversas etapas que acontecem antes da tradução do código fonte e outras que precisam ser executadas após a tradução.

A figura 1 ilustra esse conceito utilizando a *engine* V8.

Figura 1 - Fluxo de execução do Javascript na *engine* V8



fonte: Adaptado de: https://www.fhinkel.rocks/2017/08/16/Understanding-V8-s-Bytecode/

Todas as etapas que são apresentados na imagem fazem parte da cadeia de execução (*pipeline*) do javascript na *engine* V8. Conseguimos observar que a primeira etapa é fazer com que a *engine* tenha acesso ao código fonte. Após esta etapa, o código é passado para o *parser* que é responsável pela geração da árvore de sintaxe abstrata que, simplificando os termos, é uma representação do código fonte em uma estrutura sintática mais simples para o computador manipular. A árvore de sintaxe abstrata é utilizada pelo interpretador, no V8, esse componente é chamado de *Ignition*; que por sua vez gerará o *bytecode*. O *bytecode* é uma abstração do código de baixo nível, ou seja, do código final a ser executado na máquina do usuário (HINKELMANN, 2017). O *bytecode* é muitas vezes utilizado pelo compilador de otimização, *TurboFan*, para gerar um novo *bytecode* melhorado.

Essas alterações no *bytecode* só acontecem quando o código interpretado é marcado como “quente” pelo *profiler*. Sendo assim, percebemos que existem muitos

processos envolvidos executando em segundo plano para que seja possível desenvolvermos novas funcionalidades em uma página web usando javascript. E é justamente neste ponto que conseguimos começar a explicar a necessidade da existência do webAssembly.

Cada uma dessas etapas apresentadas na imagem é escalonada pela *engine*, sendo assim, em tempo de execução, acontece um pouco de análise sintática (*parsing*), depois a interpretação de algum trecho, depois um pouco de compilação dos trechos “quentes” e depois um pouco de *garbage collection*, por exemplo (CLARK, 2017). Como é a *engine* quem distribui esses processos da maneira que achar melhor, não existe uma ordem específica para a execução das etapas.

A autora ainda deixa claro que graças a esse escalonamento dinâmico o desempenho do Javascript moderno é muito superior ao que era visto no início da linguagem, contudo, mesmo com as melhorias na *engine* que possibilitaram que os desenvolvedores criassem aplicações cada vez maiores e mais complexas, todo o processo ainda continua lento se comparado às linguagens compiladas e essa lacuna é exatamente do tamanho do webassembly

2.1.4 Webassembly

Webassembly é uma nova linguagem de programação de baixo nível, performática, que pode ser executada em web browsers modernos que oferecem o suporte à tecnologia. O uso mais comum, e que provavelmente será o mais produtivo para a maioria dos desenvolvedores web, será utilizar o Webassembly como uma linguagem alvo. Isto é, um desenvolvedor poderá escrever um código em C++, por exemplo, e compilá-lo em um módulo WASM pronto para ser usado na web em velocidade próxima a nativa (MOZILLA, 2017?b).Dessa forma, o webassembly derruba a barreira da exclusividade do javascript para o desenvolvimento de aplicações web e pode abrir novos horizontes para a criação de aplicações cada vez mais poderosas e velozes.

Para que fique claro: O webassembly não substituirá o javascript, na verdade, esse não é seu objetivo, mas sim, trabalhar como uma extensão do mesmo. A ideia principal é usar o webassembly em partes da aplicação web que precisam de melhorias de desempenho.

2.2 Como o webassembly funciona

O webassembly possui um padrão de computação realmente interessante, pois seu modelo de instruções não é desenhado para uma máquina específica como outros tipos de códigos assembly, mas sim, para uma máquina virtual baseada em pilha (CLARK, 2017). Dessa forma, as instruções em baixo nível do wasm são chamadas de instruções virtuais.

Apesar de ser um formato binário, as instruções do webassembly possuem uma forma de serem visualizadas textualmente, esse formato é chamado *Webassembly Textual Format* (*WAT)*. O *WAT* utiliza a sintaxe de expressões S para representar essas instruções (MOZILLA, 2017?b). No exemplo abaixo, está a definição de uma função em webassembly que recebe dois parâmetros *p1* e *p2* dos tipos inteiro de 32 bits e faz a soma dos mesmos:

```
(func (param $p1 i32) (param $p2 i32)
  get_local $p1
  get_local $p2
  i32.add )
```

As instruções *get_local* que seguem o corpo da expressão S acima, servem para adicionar os valores recebidos como parâmetros no topo da pilha. A importância da adição desses valores na pilha é que o webassembly executa sobre uma máquina virtual baseada em pilhas, ou seja, toda e qualquer operação adiciona ou remove valores em uma pilha virtual.

No exemplo acima, a instrução *i32.add* implicitamente remove os dois primeiros valores da pilha, soma-os e o resultado é adicionado novamente no topo da estrutura. Dessa forma conseguimos perceber particularidades do webassembly: ele é estaticamente tipado e todas as suas operações são baseadas apenas em aritmética (SURMA, 2021)

2.2.1 Desempenho do WebAssembly

Por possuir tantos benefícios como a compilação, otimizações por parte do compilador, não ser alvo de interpretação e possuir sintaxe estaticamente tipada, o

webassembly pode ser erroneamente interpretado como a única e melhor saída para o desenvolvimento de aplicações web rápidas, contudo, na prática não é tão simples assim.

O webassembly não melhorará significativamente o desempenho de aplicações que fazem o seu uso esporádico ou em partes que não tem grandes processamento de dados (COHEN, 2020) isso se dá ao fato de que o javascript moderno é muito eficiente graças a todas as particularidades que apresentamos nos tópicos anteriores. Sendo assim, de acordo com Cohen, é necessário que o uso do webassembly seja justificado em cada uma das aplicações já que o desempenho não é um fator com aparentes ganhos.

Contudo, existem alguns motivos que geralmente fazem o webassembly ser mais rápido que o javascript:

O webassembly possui um arquivo compacto, e os ganhos nesse quesito podem ser melhorados com algum tipo de compactação como o *g-zip*, sendo assim, nas aplicações que fazem a utilização de arquivos wasm, normalmente existe a sensação de que as páginas carregam mais rapidamente.

O código WASM está muito mais próximo da linguagem de máquina, portanto, o processo de decodificação é mais rápido, porque otimizações são executadas no momento da compilação o que remove a necessidade da *engine* javascript fazer esse tipo de trabalho.

O WASM é estaticamente tipado e tal funcionalidade é realmente importante já que no javascript, que é dinamicamente tipado, a *engine* precisa realizar custosas operações de reconstrução do *bytecode* otimizado, quando o tipo de alguma variável muda em trechos de código executados com frequência. No webassembly esse problema não existe já que os tipos das variáveis são imutáveis.

O webassembly pode se tornar ainda mais rápido já que possui suporte às *Single Instructions Multiple Data - SIMD* e ao processamento de dados *multi-thread*, funcionalidades, que o javascript não possui suporte. E por fim, outro fator que pode ajudar na melhoria dessa velocidade é que a coleta de lixo, etapa responsável por liberar espaços de memória que não estão sendo mais utilizados, não é necessária no webassembly, pois a memória precisa ser gerenciada manualmente pelo desenvolvedor.

Tais proposições a respeito do desempenho que pode ser adquirida em uma aplicação parece o cenário ideal para o desenvolvimento do estudo de caso de

desempenho, contudo, diferentemente de Cohen, propomos um estudo de *benchmarking* mais profundo utilizando uma aplicação de uso prático e conhecidamente custosa (DE ALBUQUERQUE, M. e DE ALBUQUERQUE, M, 2000): O reconhecimento de dígitos manuscritos. Nossa proposta é desenvolver um web site que seja capaz de receber uma imagem contendo alguns números escritos à mão de um usuário e classificá-los.

A ideia central é realizar todo o pré-processamento da imagem e classificação dos dígitos em webassembly e em javascript. Para atingir os objetivos, precisamos realizar os processos no *client-side* e medir o tempo de execução dos algoritmos em cada imagem, para, por fim, comparar os resultados entre as aplicações.

2.3 Visão Computacional

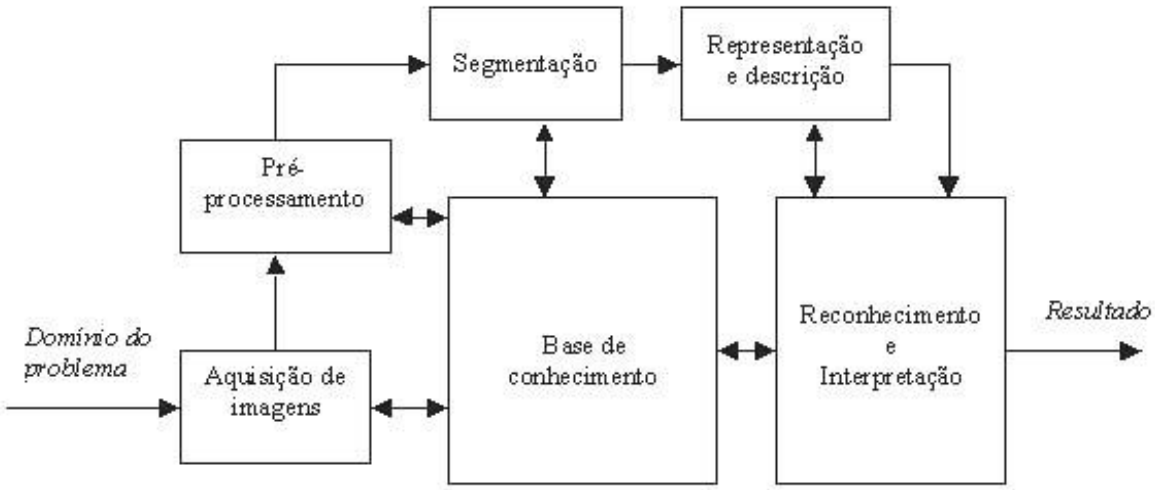
Para um melhor entendimento da visão computacional, e das etapas de processamento, se faz necessário compreender o que é uma imagem digital. Uma imagem digital pode ser definida como uma coleção de elementos discretos e de tamanho limitado distribuídos em uma matriz que representa uma paisagem. Cada membro desta matriz é conhecido como pixel, e cada um está associado a um valor que representa a cor da imagem (GONZALEZ e WOODS, 2010).

A visão computacional pode ser compreendida como uma área de estudo que tem como seu foco desenvolver e integrar às máquinas a capacidade de visão, de maneira que tenha a maior proximidade com a dos humanos. Sendo esta visão não apenas a capacidade de capturar imagens, mas também, a habilidade de, ao capturar uma imagem, processá-la e melhorá-la , tirando as partes indesejadas, entender o seu contexto e identificar os objetos de interesse (BACKES, 2019).

Também podemos encontrar a definição que coloca a visão computacional como a capacidade de extrair, a partir de uma imagem que contém objetos do mundo real, descrições significativas e relevantes (BALLARD, 1982).

Gonzalez e Woods (2010), explicam de modo geral cada etapa do processamento de imagens presente visão computacional. A figura 2 faz uma representação de todo esse processo.

Figura 2 - Etapas de um sistema de visão computacional



(fonte: Gonzalez e Woods (2000, p. 5))

Primeiro partimos de um problema que é obter informação de uma imagem, o que nos leva a primeira etapa, a aquisição de imagens. O objetivo é adquirir a imagem digital, que irá ser alvo dos demais processos.

Logo após vem o pré-processamento, aqui é feito a melhora da imagem, remoção de ruídos, aplicação de filtros, redimensionamento e etc. Tudo isso de acordo com a necessidade do problema.

A segmentação é o processo de dividir a imagem em partes que contenham os objetos de interesse. Essa é uma das etapas mais complexas da visão computacional, e se bem feita proporciona maiores chances na identificação de objetos.

Na representação e descrição temos a extração de informações quantitativas, usadas como características. Elas que irão descrever e representar os objetos, possibilitando a distinção entre eles.

No reconhecimento e interpretação, o objeto será reconhecido e/ou classificado com base nas características descobertas.

Todas essas etapas utilizam de uma base de conhecimento que é gerada com base no domínio do problema.

A constante e rápida evolução da Web fez o seu uso indispensável nos dias atuais, e também o desenvolvimento voltado para esse ambiente. A utilização da visão computacional está progressivamente mais presente em diversas aplicações e soluções Web. Como resultado, o uso de processamento de imagem, que é necessário para a visão computacional, também aumentou. No entanto, também

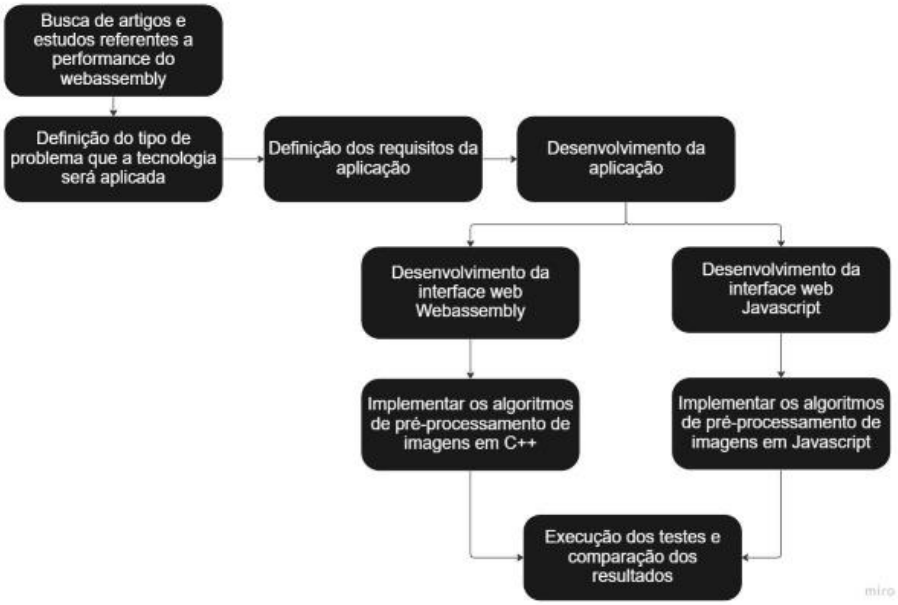
podemos encontrar seu uso para outros fins em sites que aumentam a resolução de fotos, usam filtros de imagem e entre outros lugares.

Na monografia de Cunha (2018), o Reconhecimento Óptico de Caractere é usado em duas aplicações. Servindo para comparar o processamento local e remoto e avaliar o potencial dos dispositivos móveis. É concluído que o processamento remoto não tem grandes vantagens em comparação ao local. Enfatizando a evolução dos recursos do *client-side*. E também demonstra situações onde o WASM pode aumentar as vantagens do *client-side*. E trazer novas possibilidades para o desenvolvimento Web.

3 METODOLOGIA

Neste capítulo serão apresentados os métodos utilizados para alcançar os objetivos gerais e específicos propostos neste trabalho. Todo o desenvolvimento foi dividido em etapas que estão representadas na figura 3.

Figura 3 - Fluxograma de atividades



fonte: elaboração própria

3.1 Requisitos da aplicação

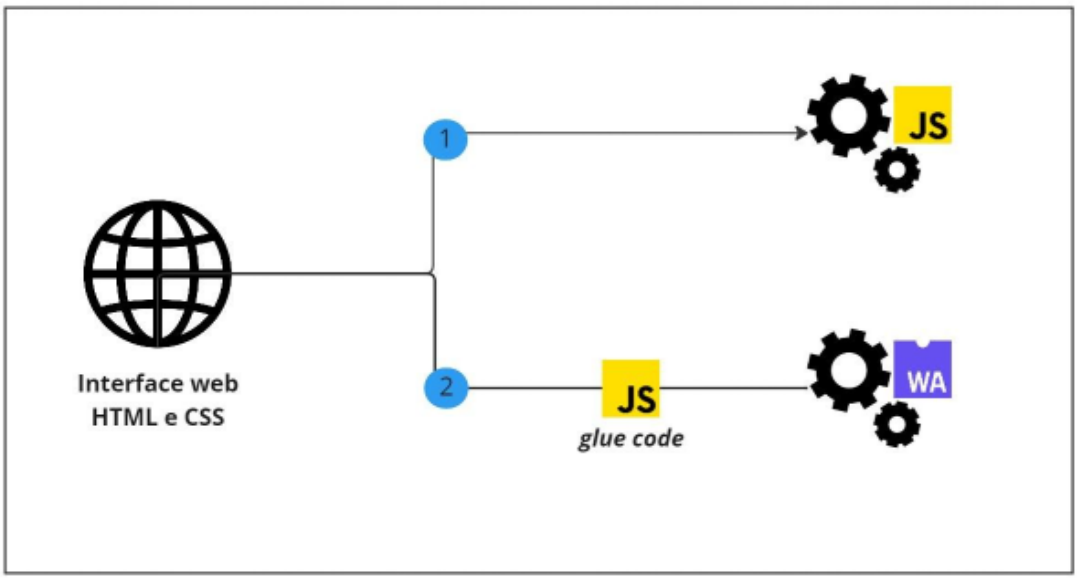
O objetivo da aplicação é fornecer uma interface web que possibilite receber imagens com um ou mais dígitos escritos à mão e classificá-los. Como foi dito anteriormente, duas versões da aplicação foram criadas para que uma comparação entre javascript e webassembly fosse possível.

Sendo utilizado para isto, as tecnologias padrão para o desenvolvimento web: HTML, CSS e javascript em ambas aplicações. Como o webassembly não foi desenvolvido para ser um substituto do javascript, a linguagem também é necessária na aplicação que faz o uso do webassembly.

A versão que faz uso do webassembly precisa do javascript para fazer a “ponte” entre o browser e o código binário WASM. Esse javascript intermediário, muitas vezes

chamado *glue code*, pode ser gerado automaticamente por um compilador no momento da compilação (EMSCRIPTEN, 2015b). Neste trabalho utilizaremos o Emscripten como ferramenta usada para compilar os códigos fonte escritos em C++ para webassembly. O *glue code* gerado pelo emscripten será usado no projeto, pois abstrai as diversas configurações necessárias para que o webassembly funcione corretamente no *browser*. A figura 4 mostra ilustra a estrutura da aplicação web descrita anteriormente.

Figura 4 - Arquitetura da aplicação



fonte: elaboração própria

A figura 4 mostra que a interface web das duas aplicações é escrita em HTML e CSS, contudo, existe uma distinção após este ponto. O ponto 1 representa a aplicação que utiliza javascript. Já no ponto 2, o que representa a aplicação que utiliza webassembly.

Entendida cada uma das particularidades das duas aplicações, o funcionamento base das mesmas é equivalente.

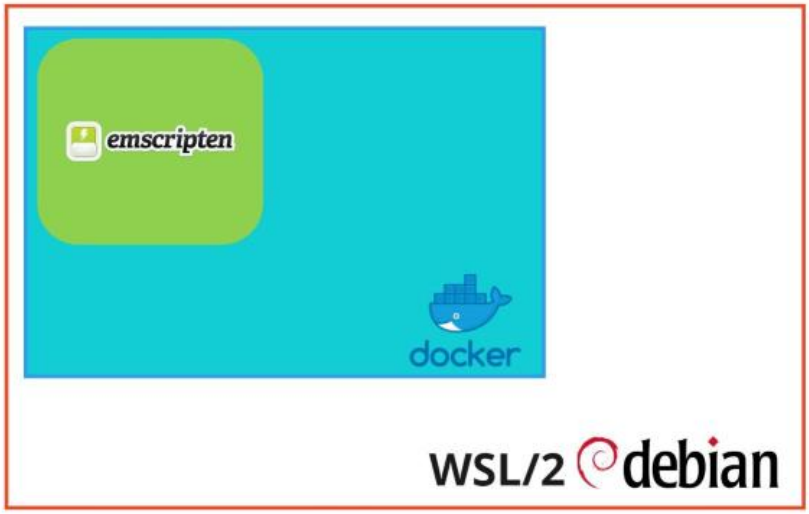
3.2 Configuração do ambiente de desenvolvimento

Todo o desenvolvimento da aplicação aconteceu no sistema operacional Debian GNU/Linux 10[8] (*WSL 2*).

[8] GNU/Debian. 2022. Disponível em: https://www.debian.org

Na aplicação desenvolvida, uma imagem oficial da toolchain Emscripten foi baixada através do Docker Hub e funciona de forma independente dentro de um container docker. A figura 5 mostra a relação entre o sistema operacional e a imagem do Emscripten.

Figura 5 - Representação do ambiente de desenvolvimento



fonte: elaboração própria

O docker[9] é uma ferramenta de virtualização para desenvolver, enviar e executar aplicativos. Basicamente ele permite que cada componente do software seja isolado e tratado como uma parte independente da infraestrutura do projeto.

3.3 Aquisição da imagem

Seguindo o fluxo ilustrado na figura 2, a aplicação desenvolvida durante este trabalho possui uma interface formada apenas pelo botão para fazer a escolha de uma imagem arbitrária do computador do usuário, com as extensões png, jpg ou jpeg; um canvas, na figura representado pelo quadrado com os cantos arredondados; e um botão de configuração com algumas opções para visualização de etapas de pré-processamento. Após a escolha da imagem, esta será desenhada na tela para visualização no canvas. Um canvas é uma elemento do HTML5 que possibilita o desenho e manipulação de imagens no browser via javascript (WHATWG, 2022). A figura 6 é a interface principal do sistema.

[9] Docker. 2022. Disponível em: https://www.docker.com/

Figura 6 - Layout da aplicação

fonte: elaboração própria

3.4 Pré-processamento da imagem

O pré-processamento é uma etapa que visa corrigir possíveis defeitos na imagem. Na aplicação desenvolvida, são realizadas operações de redimensionamento. Além disso, etapas de espelhamento e rotação da imagem são necessárias para adequá-la à rede neural.

Para reduzir a complexidade do pré-processamento, primeiro a imagem foi convertida para tons cinza.

3.4.1 Escala de tons de cinza

Converter imagens para tons de cinza ou *grayscale* é uma das técnicas de processamento mais utilizadas onde a ideia principal é a planificação da imagem.

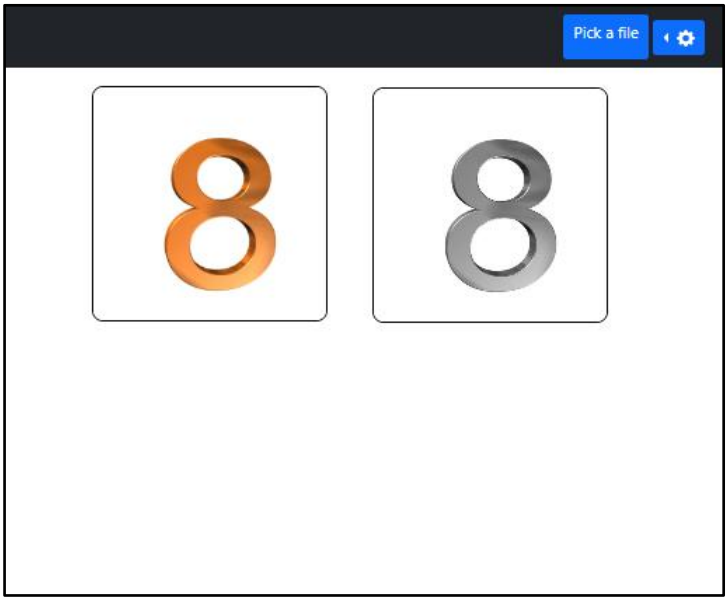
Para isso acontecer o valor dos seus pixels em RGB, que tem 24 bits, será convertido em 8 bits. Com isso os pixels terão os seus valores variando entre 0 e 255, cujo resultado será a imagem em *grayscale* (SARAVAN, 2010).

A forma escolhida para o desenvolvimento deste projeto se chama coloquialmente de luma e está baseada na recomendação BT.709 para transmissão de imagens digitais (ITU, 2002). A transformação em tons de cinza usando este método consiste em gerar um novo valor para o pixel resultante a partir de uma soma ponderada dos valores de *RGB* do pixel original. A fórmula para isso é a seguinte:

$$Y = R * 0.299 + G * 0.587 + B * 0.114$$

Onde Y é o pixel de saída. Para a aplicação do método, foi escrita uma função que recebe os valores dos pixels e aplica a fórmula acima a cada um deles. A imagem abaixo mostra o resultado do método.

Figura 7 - Aplicação dos tons de cinza utilizando o método luma



fonte: elaboração própria

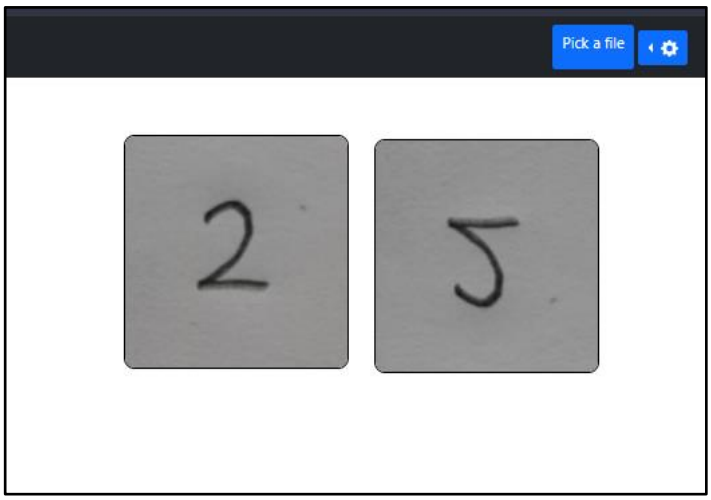
No lado esquerdo da imagem, observa-se uma ilustração colorida desenhada no canvas, já no lado direito, a mesma ilustração consegue ser vista, porém já padronizada em tons de cinza.

3.4.2 Espelhamento da imagem

Para alimentar a rede neural com as informações dos pixels da imagem precisamos refleti-la. Esta etapa do pré-processamento é necessariamente uma particularidade da rede neural escolhida para a classificação. Durante os testes, foi percebido que a rede neural não estava classificando os dígitos de maneira correta na maioria dos casos. Este fato ia de encontro com a precisão apresentada por Eichner (EICHNER, 2014), desenvolvedor da rede neural utilizada. Esta foi uma das principais dificuldades enfrentadas no pré-processamento, pois o autor não explica a presença da etapa no periódico. Sendo assim, o código precisou ser analisado e comparado com os resultados que Eichner apresentou para, dessa forma, descobrirmos a necessidade do espelhamento.

Foi criada uma função de transformação matricial para o espelhamento da imagem. Nesta função, cada um dos valores dos pixels recebe uma nova posição calculada pelo algoritmo de espelhamento. Como resultado, conseguimos uma imagem espelhada verticalmente.

Figura 8 - Aplicação do algoritmo de espelhamento



fonte: elaboração própria

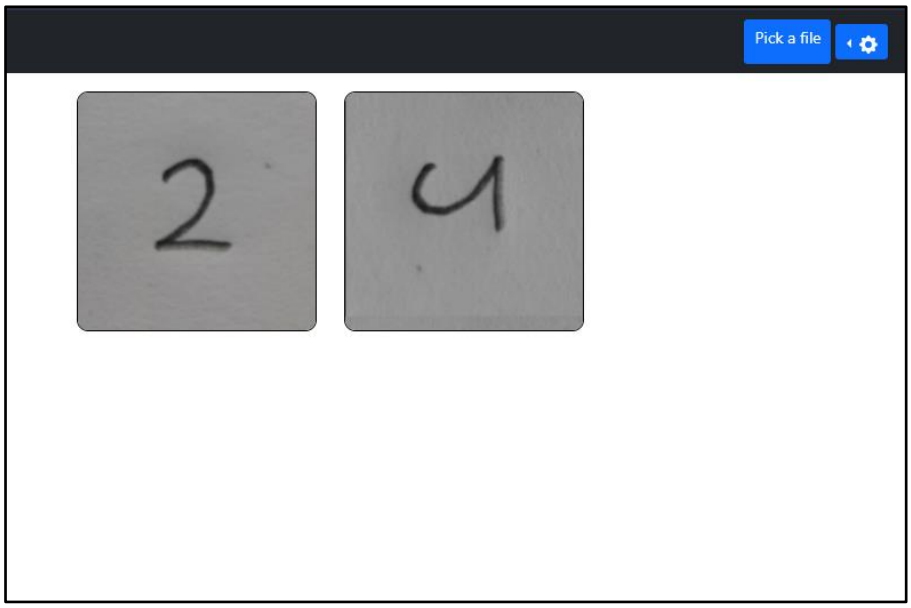
No exemplo acima foi utilizada uma foto com um dígito manuscrito. Do lado esquerdo a foto original e do lado direito, o resultado após a imagem passar pelo algoritmo de espelhamento vertical.

3.4.3 Rotação da imagem

Também na etapa de pré-processamento, a imagem precisa ser rotacionada em 90 graus no sentido horário. Essa característica, assim como a descrita no tópico anterior, não está documentada na descrição da rede neural, portanto, apenas foi descoberta sua necessidade, analisando minuciosamente a aplicação e comparando o comportamento da rede neural já implementada.

Na figura 9 podemos ver o resultado obtido após a rotação de 90 graus no sentido horário.

Figura 9 - Aplicação do algoritmo de rotação da imagem



fonte: elaboração própria

3.4.4 Redimensionamento da imagem

Esta é a última etapa do pré-processamento em nosso projeto. A rede neural será alimentada com os valores dos pixels da imagem resultante, contudo, a rede classificadora tem um número específico de parâmetros de entrada. A rede neural utilizada possui 784 *inputs* de entrada. A quantidade de neurônios de entrada é explicada pelo fato da rede neural ser treinada utilizando o dataset MNIST (LECUN et al, 1998) que define as imagens em dimensões de 28x28 pixels.

E isto pode ser um problema, visto que a porção da imagem que contém o dígito pode ser superior a esta dimensão, sendo assim, uma imagem com área diferente a esta deve ser redimensionada antes de ser passada para alimentar a rede neural classificadora.

O método implementado para o redimensionamento da região de interesse da imagem é chamado de *nearest neighbor interpolation* (PARKKINEN et al., 2009). O algoritmo para a redução de escala utilizando este método basicamente seleciona um número X de pixels de entrada para serem utilizados como correspondência a um único pixel de saída. O redimensionamento, apesar de estar na etapa de pré-processamento, só acontecerá após a segmentação para evitar a perda de informações da imagem.

3.5 Segmentação

Para o processo de segmentação, a imagem precisa ser divida em segmentos ou regiões de interesse. Basicamente para a aplicação desenvolvida ao longo deste trabalho, a única região de interesse nas imagens é a que possui os dígitos manuscritos, dessa forma, nosso principal objetivo nesta etapa é definir a localização dos dígitos na imagem. Para conseguir atingir os objetivos da etapa de detecção das regiões de interesse, dividimos o processamento em passos que serão descritos nos subtópicos a seguir.

3.5.1 Binarização

A estratégia para identificar regiões de interesse na imagem começa com a binarização, que divide a imagem em tons de cinza em dois planos: fundo e objeto. Um valor limiar T é escolhido para definir os pixels que serão identificados como brancos ou pretos, valores acima de T definem pixels brancos e abaixo, pretos.

É interessante notar que os algoritmos de limiarização possuem diferenças entre si e estas diferenças estão, principalmente, na ideia de seleção deste limiar. Quando o limiar T for constante, ou seja, aplicado para binarizar toda imagem, tem seu processo de binarização chamado de limiarização global. Por conseguinte, a definição de um algoritmo onde um limiar T é variável, é chamado de binarização variável, ou local (GONZALEZ e WOODS, 2010).

O principal problema na geração da imagem binária é determinar um limiar automaticamente que seja capaz de dividir os pixels de maneira aceitável. Para o escopo deste trabalho, que se limita ao estudo de desempenho de execução das implementações, uma técnica clássica de um limiar global poderia ser utilizada. Sendo assim, o método de Otsu (OTSU, 1979) foi escolhido.

A técnica de Otsu funcionará bem para nossos experimentos, pois, apesar de ser uma técnica mais simples, as imagens que utilizaremos como entrada na aplicação, também possuem *backgrounds* simples. O que resolve nosso problema da binarização sem adição de mais complexidades.

O método implementado na aplicação funciona buscando um limiar que diminua a variância intra-classe, isto é, o método buscará iterativamente um limiar onde a soma ponderada das variâncias entre o *background* e o *foreground* seja a

menor possível (OTSU, 1979). A fórmula abaixo descreve o comportamento do método.

$$\sigma_w^2(t) = w0(t) \cdot \sigma_0^2 + w1(t) \cdot \sigma_1^2$$

O autor explica a fórmula demonstrando que $w0(t)$ e $w1(t)$ são as probabilidades das duas classes, plano de fundo e plano principal, divididas pelo limiar t cujo valor está entre 0 e 255 inclusivamente.

Assim como as implementações expostas nos pontos anteriores, a implementação do método de Otsu para a definição de um limiar global não foi diferente: definimos uma função que recebe a matriz de pixels da imagem já em tons de cinza e nos retorna um valor usado para binarizar a imagem. O resultado da binarização é mostrado na figura 10.

Figura 10 - Aplicação do algoritmo de binarização utilizando o método de Otsu.

fonte: elaboração própria

Após passarmos a matriz da imagem já em escala de cinza para a função que implementa o método de Otsu, utilizamos o valor retornado para binarizar a imagem em duas cores. Na figura 10, percebemos que também foi considerado como *foreground* um pequeno ponto de ruído que não faz parte do traçado do dígito. Para resolver esse problema, foi escrito um código que desconsidera objetos muito pequenos nas etapas posteriores à binarização para evitar classificações incorretas.

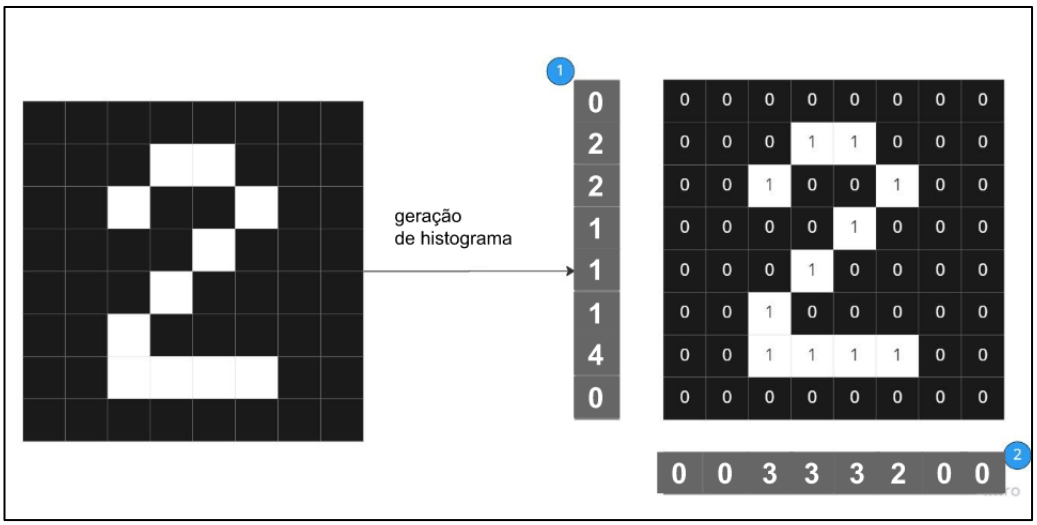
3.5.2 Definição da Região de Interesse (*ROI*)

Para a descoberta da região de interesse da imagem já binarizada, podemos utilizar a técnica de projeção de histograma. Esse método nos permitirá encontrar pontos de início e finais de objetos no *foreground* da imagem (GATTAL e YOUCEF, 2012).

A ideia por trás desse tipo de segmentação é simples: Utilizaremos a imagem binarizada para iniciar uma varredura horizontal e vertical a fim de somar a quantidade de pixels de *foreground* em cada linha e coluna. Os produtos resultantes dessas varreduras serão duas listas com os valores que nos permitirão encontrar regiões de interesse na imagem.

A figura 11 mostra, em um exemplo, a representação em memória de uma imagem binarizada e listas que representam seus histogramas.

Figura 11 - Geração de histogramas a partir de imagens binárias.



fonte: elaboração própria

Na figura 11, os dois *arrays* indicados pelos números 1 e 2 representam a soma de valores de *foreground*, ou seja, pixels de valor 1, em cada uma das linhas e colunas, respectivamente. Esses arrays podem ser desenhados em um canvas onde cada um dos elementos, representam a altura de linhas e, com isto, obtermos representações visuais de histogramas como mostra a figura 12.

Figura 12 - Representação gráfica dos histogramas.

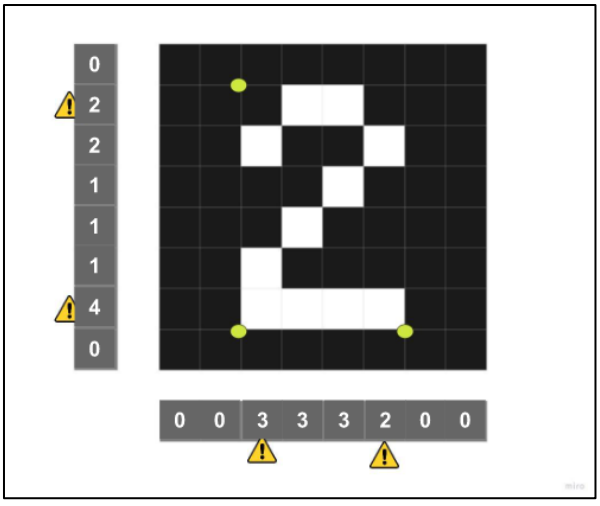
fonte: elaboração própria

No projeto desenvolvido, as funções escritas para geração dos projeções, recebem a matriz de pixels da imagem binarizada e retornam apenas duas listas com os valores da somatória dos pixels de linhas e colunas. Dessa forma, as representações visuais dessas projeções mostrados na figura 12 não possuem uso prático além da visualização. Os valores resultantes serão utilizados como uma estratégia para a definição do objetivo de descobrir a região de interesse, meta principal da nossa etapa de segmentação.

A ideia principal foi definir funções responsáveis por fazer varreduras nos histogramas gerados, e com isso, definir os pontos onde são detectadas mudanças abruptas nos valores presentes nos histogramas. Essas mudanças são onde se iniciam e finalizam as inscrições dos dígitos na imagem.

A figura 13 mostra as mudanças nos histogramas e como essas mudanças nos geram informações para definir a localização do número.

Figura 13 - Detecção de mudanças nos histogramas da imagem binária.

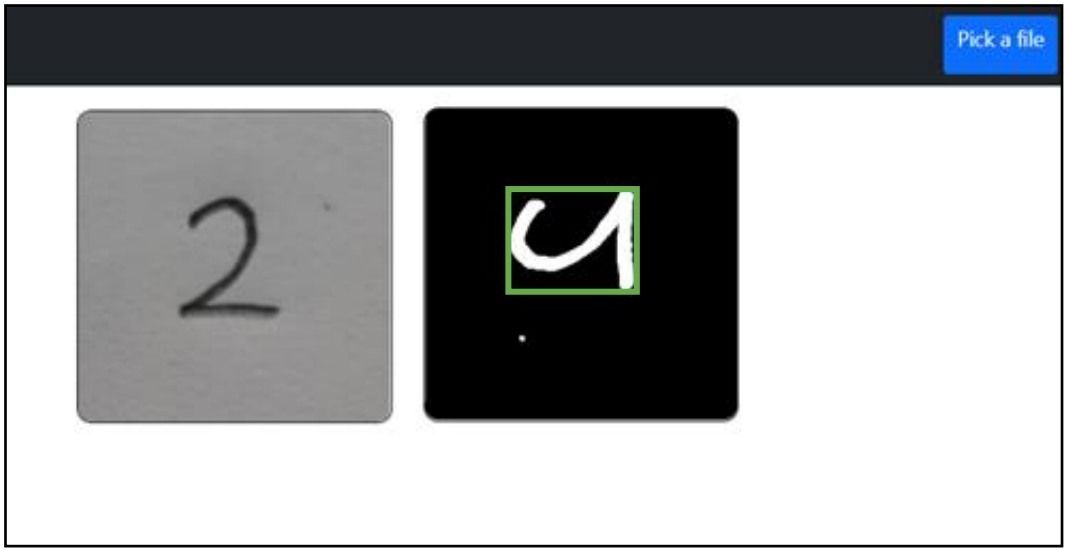


fonte: elaboração própria

Por meio de uma varredura em cada um dos histogramas, conseguimos verificar mudanças significativas nos valores em determinados momentos. No exemplo da figura 13, o histograma horizontal tem os primeiros valores zerados, e, quando o algoritmo se depara com o terceiro valor, que é diferente, essa posição é marcada como uma alteração de sequência (exclamações). As alterações na sequência dos valores dos histogramas são convertidas em coordenadas, representadas pelos pontos em verde, e utilizadas para demarcar o início e o final dos dígitos na imagem. Após a definição das coordenadas, conseguimos descartar da imagem todo o restante que não nos interessa mais.

A figura 14 mostra o funcionamento do algoritmo descrito na aplicação desenvolvida.

Figura 14 - Exemplo da experiência após a detecção da região de interesse.



fonte: elaboração própria

O retângulo em verde desenhado sobre a imagem binária, representa a região mais importante da matriz, descartando todo o restante que não é útil, antes de realizarmos a passagem para a rede neural classificadora. Nota-se que, apesar do ruído ainda estar presente na imagem binarizada, ele não foi considerado uma região de interesse. Isso se deve ao fato de que ao escrevermos o algoritmo para a varredura dos histogramas, desconsideramos áreas muito pequenas.

3.6 Classificação

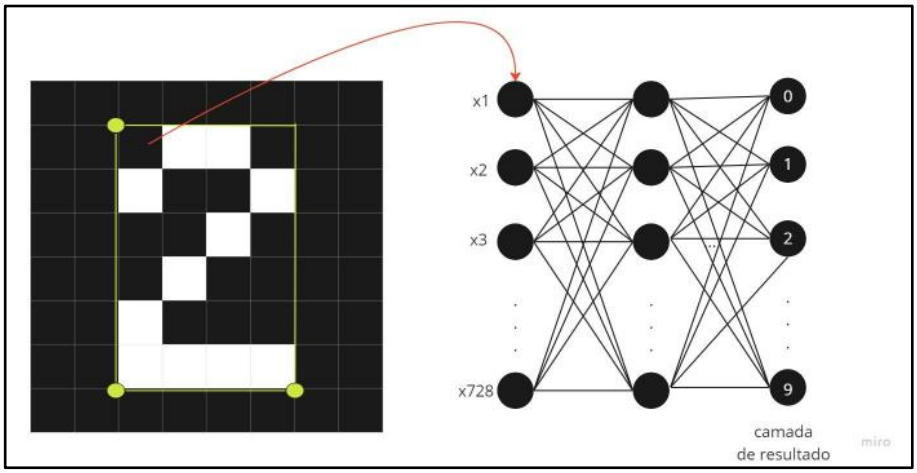
A última etapa após o pré-processamento e segmentação da imagem é a classificação. Utilizamos uma rede neural *open source* criada para testar a velocidade de execução da classificação de dígitos no *browser* utilizando javascript. Foge do escopo deste experimento, a codificação e treinamento de uma rede neural, visto que nosso objetivo é comparar o desempenho de execução de tarefas custosas no navegador em aplicações que utilizam javascript e webassembly.

A rede neural utilizada é um de um tipo específico chamado convolucional (*CNN*) que é especializada em processar dados no formato de matriz, como uma imagem (GOODFELLOW et al, 2016). Uma rede neural convolucional implementada em javascript é perfeita para nossos casos de uso, pois a partir disso, foi muito simples, adaptar o código para C++ e compilá-lo para webassembly.

O projeto *open source* do qual isolamos a rede neural, desenvolvido por Eichner, é pensado para ser utilizado isoladamente na web sem possibilidade de escolha de imagens com dígitos manuscritos. Em seu blog, o projeto foi aplicado utilizando imagens de dígitos onde o próprio usuário desenha na tela do dispositivo. Sendo assim, a ideia de isolar a rede neural classificadora, e utilizá-la em um sistema mais genérico que aceitasse imagens vindas de qualquer fonte é original e exclusivamente desenvolvida para os testes realizados neste trabalho.

A ideia principal é simplesmente lançar os pixels da região de interesse descrito no tópico anterior, como entrada para a rede neural classificadora. A rede utilizará os valores dos pixels para definir uma probabilidade de a imagem conter alguns dos números de 0 a 9. A figura 15 ilustra o processo de classificação.

Figura 15 - Utilização da *ROI* como entrada de dados para a rede neural.

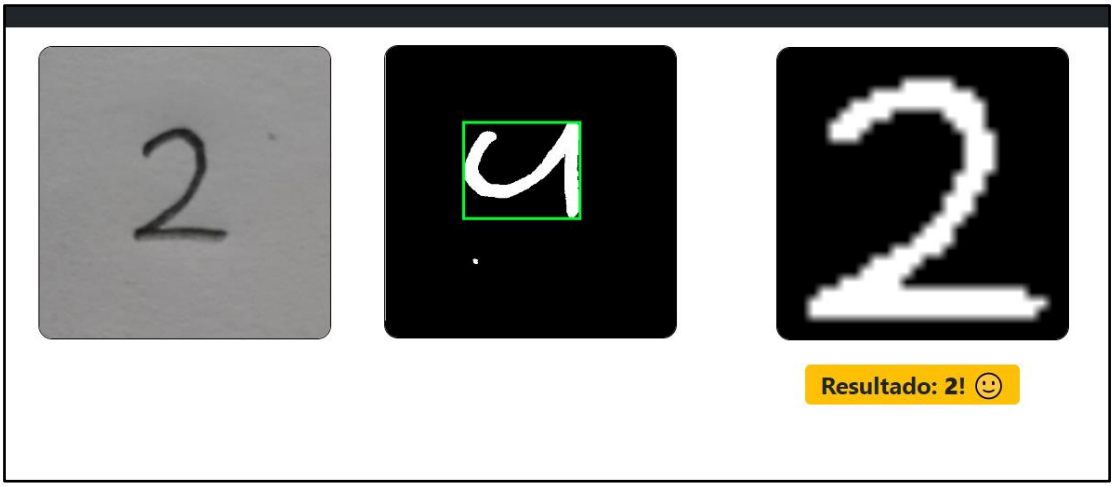


fonte: elaboração própria

Nesta etapa, a região de interesse, marcada em verde, está padronizada no tamanho de 28x28, exatamente a quantidade de entradas que a rede neural possui. Na figura 15, as entradas estão representadas por valores que vão de x1 até x728. A última camada da rede neural será a que determina o resultado. Esse resultado ficará distribuído em porcentagens que indicarão o nível de probabilidade do número ser o indicado.

Uma rede neural convolucional possui diversos detalhes de seu funcionamento que foram omitidos ao longo deste tópico visando a simplificação do escopo do trabalho. Contudo, o objetivo principal, de processar a imagem e classificar o dígito, foi atingido nas aplicações em javascript e webassembly.

Figura 16 - Classificação do dígito da região de interesse.



fonte: elaboração própria

A figura 16 mostra o resultado final após a classificação. O alerta em amarelo é lançado na tela do usuário com o resultado final.

4 RESULTADOS

Para chegar ao resultado final apresentado neste capítulo, as aplicações foram executadas em um notebook Dell Inspiron 3584, com o processador Intel Core i3-7020U CPU de 2,30 GHz de 64 bits em um ambiente controlado de 18º *celsius*. O notebook possui 16GB de memória RAM e 500 GB de SSD e está executando o sistema operacional Microsoft Windows 10.

Para os testes de benchmarking, um dataset de imagens com dígitos manuscritos foi utilizado. O conjunto de dados possui 14119 imagens. Esse dataset foi gerado durante o desenvolvimento deste trabalho, pois precisávamos de imagens com dígitos no estilo MNIST (LECUN et al, 1998), com mais de um dígito. Então a ideia foi gerar um novo dataset sintético, utilizando como base o conjunto MNIST, com imagens possuindo até 7 dígitos. Sendo assim, as 14119 imagens foram divididas em grupos de 2017 elementos, cada grupo possuindo uma quantidade de dígitos variando de 1 a 7. As dimensões das imagens também variam gradativamente entre 200 x 200 até 1400 x 1400 pixels. Isso significa que as imagens têm dimensões proporcionais a sua quantidade de dígitos.

Abaixo são mostradas algumas imagens que foram utilizadas como entrada nas aplicações desenvolvidas.

Figura 17 - Imagens 200x200 contendo um único dígito.

fonte: elaboração própria

Figura 18 - Imagens contendo três, cinco e sete dígitos.

fonte: elaboração própria

O processamento e classificação de cada uma dessas imagens foi executado nos seguintes *browsers* recém instalados*:* Google Chrome na versão 108.0.5359.100, Brave na versão 1.46.144, Microsoft Edge na versão 108.0.1462.46 e Mozilla Firefox na versão 109.0.1.

Para a medição dos tempos de execução, utilizamos a função *performance.now()* do Javascript que foi chamada no inicio e no final da execução de cada um dos métodos de pré-processamento e classificação utilizados. Os resultados são subtraídos, obtendo-se assim o tempo de execução com precisão de microssegundos (MOZILLA, 2015?). Ao final, todos os *timestamps* gerados foram somados para obtermos os resultados listados na tabela a seguir:

Tabela 1 – tempo total da execução dos testes tomado por cada um dos navegadores em cada uma das tecnologias.

Browsers	Tecnologias	
	Javascript (horas)	Webassembly (horas)
Brave	6,79	4,03
Firefox	3,49	4.36
Edge	7,83	3,02
Chrome	7,75	3,11

Fonte: elaboração própria

Apesar do javascript possuir diversos mecanismos de otimização, o webassembly realmente conseguiu performar melhor na maioria dos navegadores. O caso de maior destaque foi no Microsoft Edge onde a média de execução do webassembly foi 317% superior. Apesar do sucesso no Edge, um caso interessante aconteceu no Mozilla Firefox onde, além de obter a melhor pontuação na medição de desempenho usando Javascript, a versão utilizando webassembly performou pior, entregando resultados cerca de 41% inferiores.

4.1 Comparações

Para criação dos gráficos e análise dos resultados, agregamos os resultados do tempo de execução de cada imagem nas informações contidas nas tabelas 1 e 2.

Tabela 2 - **Cálculos dos resultados do tempo de execução (em milissegundos) do código JS, em cada navegador**

Medidas	Brave	Firefox	Edge	Chrome
Média	2.672,37	1.043,64	3.969,52	2.768,68
Desvio padrão	2.357,03	830,83	5.948,11	2.407,17
Valor mínimo	78	55	80,3	85,9
Valor máximo	36.453	4.293	85.911,20	16.333,60

Fonte: elaboração própria

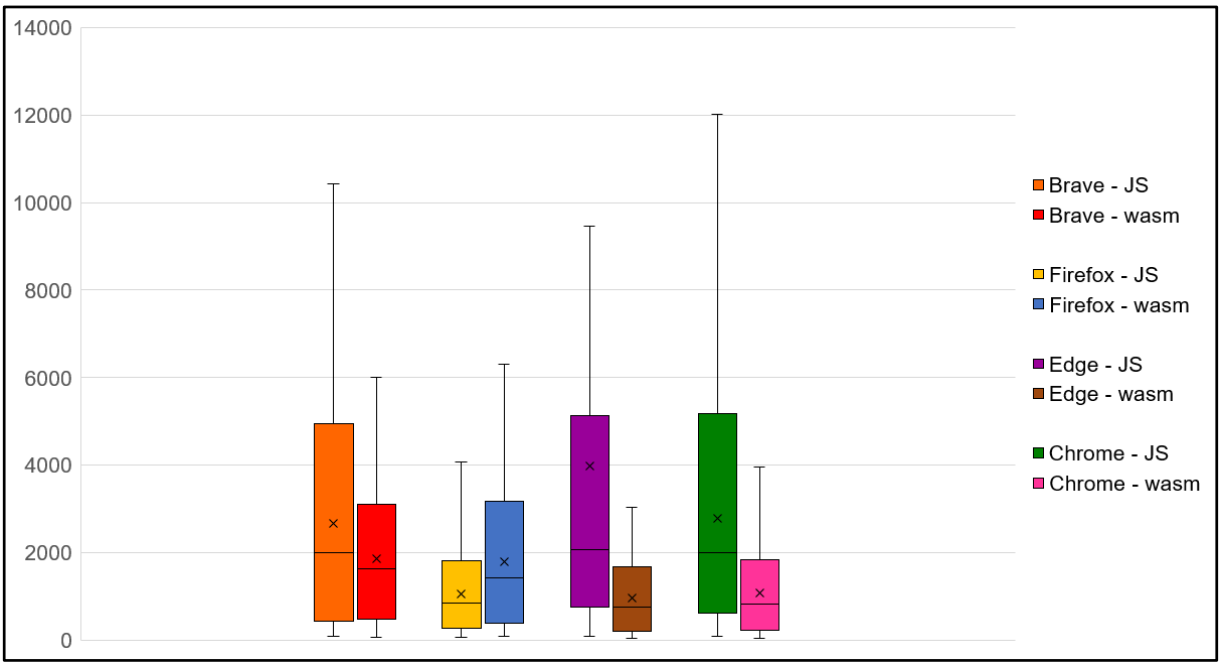
Tabela 3 - **Cálculos dos resultados do tempo de execução (em milissegundos) do código WASM, em cada navegador**

Medidas	Brave	Firefox	Edge	Chrome
Média	1.846,29	1.791,66	1.067,98	949,87
Desvio padrão	1.415,40	1.455,59	902,95	766,23
Valor mínimo	50	86	46,90	42,60
Valor máximo	6.009	6.316	23.114,90	3.912,10

Fonte: elaboração própria

A figura 17 mostra um gráfico do tipo boxplot para a comparação do tempo de processamento dos testes em cada um dos navegadores. A escala a esquerda representa a quantidade de tempo em milissegundos utilizada pela aplicação no pré-processamento, segmentação e classificação de uma imagem com um ou mais dígitos.

Figura 19 - variância do tempo (milissegundos) de processamento entre navegadores



fonte: elaboração própria

O *boxplot* na figura 17 nos mostra uma execução significativamente superior dos algoritmos implementados em webassembly. Nos navegadores Brave, Chrome e Edge a execução média (indicada por um “x” nas caixas) foi, respectivamente, 44%, 159% e 317% superior em comparação com o javascript. O que é de fato uma questão muito interessante já que ambos os *browsers* utilizam o mesmo motor baseado no Chromium, ou seja, todos estes softwares utilizam a *engine* V8 explicada no referencial teórico do trabalho.

Analisando o desempenho do Mozilla Firefox percebemos que ele teve um desempenho excelente na aplicação que utiliza javascript e isto, não se refletiu nos testes em webassembly, como foi o padrão visto nos outros browsers. A média de execução piorou cerca de 41% em relação ao javascript.

O Firefox utiliza um motor de javascript diferente dos outros navegadores utilizados neste teste, chamado *SpiderMonkey* [10] e percebemos pela análise da caixa correspondente ao *browser* que a implementação do *JIT* deste motor é mais eficiente na execução do javascript se comparada ao V8. O que pode ser informação importante em um ambiente de desenvolvimento comercial, se considerarmos que a maioria dos usuários de determinada aplicação utilizam o firefox como navegador principal.

Notamos, pelo intervalo interquartílico (tamanho da caixa), que a dispersão dos valores é muito menor na aplicação webassembly. Ou seja, o tempo de processamento da aplicação em webassembly é mais estável. Essa informação pode ser importante em aplicações onde a previsibilidade é um fator importante.

Toda a variação dos resultados no Chrome, Brave e Edge na aplicação em javascript pode ainda ter sido atenuada visto o funcionamento da *engine V8.* Já que a execução dos testes aconteceu de maneira sequencial e sem atualização da página, o *JIT* pode ter conseguido atuar em certos momentos, otimizando o código fonte das funções de execução repetitiva. Esse dado é importante de ser ponderado, pois aplicações comerciais normalmente não são executadas exaustivamente nos computadores de usuários.

[10] SpiderMonkey. 2023. Disponível em: https://spidermonkey.dev/

5 CONSIDERAÇÕES

Neste trabalho foram desenvolvidas duas aplicações web que podem usar imagens de fontes externas para o pré-processamento, segmentação e classificação de dígitos manuscritos. A principal hipótese para desenvolvimento deste experimento é que o processamento de imagens é de fato uma tarefa computacionalmente custosa. E esse motivo foi um caso de uso perfeito para testar o webassembly: uma tecnologia promissora de baixo nível que promete execução mais veloz no *browser* em relação a linguagem predominante no *client-side*, o javascript.

A aplicação tem requisitos simples: o usuário deve fazer o upload de uma imagem contendo dígitos manuscritos e o sistema será responsável por processar essa imagem, segmentá-la para descobrir a localização dos dígitos e por fim, classificá-los com uma rede neural em valores de 0 a 9. Todo o processo deve acontecer no *client-side.* Tudo inteiramente na máquina do usuário.

Sendo assim, utilizar o poder computacional dos computadores dos usuários, para realizar tarefas computacionalmente complexas pode ser uma boa ideia, visto a evolução do *hardware* presentes nessas máquinas.

O processamento de imagens faz parte do campo de estudo chamado visão computacional. Neste trabalho, todos os algoritmos de processamento de imagens descritos no capítulo 3 foram escritos em javascript e C++ (compilado para webassembly). Sendo assim, todas as etapas que envolvem a preparação e segmentação da imagem foram implementadas ao longo deste projeto.

No processo de reconhecimento dos dígitos presentes na imagem, utilizamos uma rede neural convolucional de um projeto *open-source*. Então, realizamos a tarefa de isolar a rede neural classificadora já treinada para adaptá-la a nosso favor, classificando imagens do mundo real, após o processamento.

Depois do isolamento da rede neural, e validação de nossa proposta, os esforços foram focados na criação de duas aplicações a fim de executarmos testes de benchmarking e validar nossas hipóteses em relação a superioridade do webassembly.

É importante ressaltar que o trabalho não endossa de nenhuma forma a proposição de que o webassembly substituirá o javascript em algum momento da história. Pelo contrário, concordamos que o uso do webassembly no *browser* só é possível com a utilização do javascript.

Com a execução dos testes sobre um dataset sintético do tipo MNIST de 14119 imagens, concluímos que o desempenho em relação a velocidade de execução do webassembly, nas condições descritas neste trabalho, pode superior à do javascript na maioria dos navegadores. Isso foi demonstrado pelo gráfico presente no tópico 4.1 onde o webassembly conseguiu atingir valores de até 317% melhores que o javascript, como no caso do Microsoft Edge.

O Mozilla Firefox apresentou dados de um desempenho extremamente estável na aplicação que utiliza javascript ao contrário dos demais navegadores em questão. A velocidade de execução do javascript conseguiu ser 41% superior em relação ao webassembly neste navegador. Sendo assim, aplicações comerciais onde a maioria dos clientes utilizam este *browser* podem utilizar isto ao seu favor, sem a necessidade de implementação de código em webassembly.

Durante os testes, as aplicações foram alimentadas com as imagens sintéticas do dataset e a cada 2017 imagens, acrescentamos mais um dígito e aumentamos a dimensão das imagens em 200 pixels na altura e na largura. Esta técnica visa testar a otimização das *engines* javascript mediante a um código repetitivamente executado que sofre alterações bruscas. Isso significa que o resultado dos navegadores na execução do código em javascript, no mundo comercial, poderia ser pior já que não haveria tempo suficiente para otimizações.

Na execução dos testes, as imagens eram passadas para a aplicação de maneira sequencial e isso possibilitou a otimização de velocidade de execução via *JIT* durante o processamento de cada grupo de imagem na aplicação em javascript.

Na metodologia escolhida, a melhoria de velocidade ganha na migração do javascript para o webassembly pode ser significativa ao usuário final. Ao analisarmos o gráfico da figura 17, percebemos que o webassembly, na maioria das vezes, entrega resultados expressivamente mais consistentes, e isso pode ser um fator interessante no desenvolvimento de aplicações onde a previsibilidade de velocidade da execução importa.

A metodologia apresentada foi suficiente para testarmos e avaliar se a hipótese da superioridade de aplicações que utilizam webassembly, era verdadeira. As etapas de pré-processamento e segmentação realmente não são comuns de serem vistas executando no *client-side* por serem custosas demais. As duas aplicações foram desenvolvidas com o máximo de cuidado para funcionarem de maneira equivalentes apesar das diferenças claras entre C++ (compilada para webassembly) e javascript.

Por fim, a utilização do webassembly ainda é muito complexa. Existem documentações extensas a respeito do funcionamento da tecnologia, contudo, no uso prático, as mensagens de erros, configurações necessárias e entendimento de conceitos não são triviais.

Acreditamos no futuro promissor do webassembly. A tecnologia continua em desenvolvimento, mas já entrega resultados valiosos para grandes negócios e concluímos que se a tecnologia não for estritamente necessária para um pequeno projeto, o uso da mesma pode trazer uma complexidade desnecessária para o desenvolvimento. Caso contrário, o uso do webassembly é valioso, principalmente para aplicações robustas que prezam pelo desempenho e estabilidade, como o Figma[11], Autocad[12] e Unity[13] que já o utilizam.

[11] Figma. 2023. Disponível em: https://www.figma.com/

[12] Autocad. 2023. Disponível em: https://www.autodesk.com.br/products/autocad-web

[13] Unity. 2023. Disponível em: https://unity.com/pt

REFERÊNCIAS

ANDREASEN, E. et al. **A Survey of Dynamic Analysis and Test Generation for JavaScript. ACM Comput**. ACM Computing Surveys, 2017.

BACKES, A. R.; JUNIOR, J. J. d. M. S. **Introdução à visão computacional usando Matlab**. [S.l.]: Alta Books Editora, 2019.

BALLARD, D. H.; BROWN, C. M. **Computer vision**. [S.l.]: Prentice Hall, 1982.

BODILY, Samuel. (2009). **Browser Wars: Microsoft Versus Netscape**. Darden Business Publishing Cases. 1. 10.1108/case.darden.2016.000053.

BRINKMANN, Ron. **The Art and Science of Digital Compositing: Techniques for Visual Effects, Animations and Motion Graphics.** 2 edição. San Francisco: Morgan Kaufmann Publishers. 2008.

CLARK, Lin. **A crash course in Just-in-Time (JIT) compilers.** [S.l.].2017. Mozilla Hacks. Disponível em: <https://hacks.mozilla.org/2017/02/a-crash-course-in- just-in-time-jit-compilers>. Acesso em: 23 de fev. 2022.

CLARK, Lin. **Creating and working with webassembly modules.** [S.l.]. 2017. Mozilla Hacks. Disponível em: <https://hacks.mozilla.org/2017/02/creating-and-working-with-webas sembly-modules/>. Acesso em: 29 de jun. 2022.

CLARK, Lin. **What makes webassembly fast?.** [S.l.]. 2017. Mozilla Hacks. Disponível em: <https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/>. Acesso em: 23 de fev. 2022.

COHEN, Omri. **Go webassembly performance benchmark.** [S.l.]. 2020. Code the Cloud. Disponível em: <https://dev.bitolog.com/go-webassembly-performance-benchmark/>. Acesso em: 30 de jun. 2022.

CUNHA, Luiz Henrique da Silva. **Avaliação de desempenho de visão computacional em aplicações móveis**. 2018. 60 f. Monografia (Graduação em Engenharia da Computação) – Instituto de Ciências Exatas e Aplicadas, Universidade Federal de Ouro Preto, João Monlevade, 2018.

DE ALBUQUERQUE, Márcio Portes; DE ALBUQUERQUE, Marcelo Portes. **Processamento de imagens: métodos e análises**. Rio de Janeiro, Brasil, v. 12, 2000.

EICHNER, Hubert. 2014. MYSELPH. Disponível em: <http://myselph.de/neuralNet.html>. Acesso em 23 de fev. 2022.

EMSCRIPTEN, Contributors. **Emscripten compiler toolchain.** [S.l.]. 2015 Emscripten. Disponível em: <https://emscripten.org/> Acesso em: 25 de jun. 2022.

EMSCRIPTEN. **Building to Webassembly.** [S.l.] 2015. Emscripten. Disponível em: <https://emscripten.org/docs/compiling/WebAssembly.html#webassembly>. Acesso em 24 de ago. de 2022.

GATTALl, Abdeljalil & CHIBANI, Youcef. **Combining Multiple Segmentation Methods for Handwritten Digit Recognition of Algerian Bank Checks.** 2nd International Conference on Information Systems and Technologie, 2012.

GONZALEZ, R. C.; WOODS, R. C. **Processamento digital de imagens**. Pearson Prentice Hall, 2010.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. **Deep Learning (Adaptive Computation and Machine Learning series).** Massachusetts Institute of Technology Press. 2016.

HINKELMANN, Franziska. **Understanding V8's bytecode.** [S.l.]. 2017. Franziska Hinkelmann. Disponível em: <https://www.fhinkel.rocks/2017/08/16/Understanding -V8-s-Bytecode> Acesso em: 25 de jun. 2022.

ITU, ITURBT. **Parameter values for the HDTV standards for production and international programme exchange.** *Recommendation ITU-R BT* 709-5. 2002.

JANETAKIS, Nick. **Was web development better back in the early 2000s?.** [S.l.]. 2018. Nick Janetakis. Disponível em: <https://nickjanetakis.com/blog/was-web-development-better -back -in-the-early-2000s>. Acesso em: 16 de jan. 2022.

KWAME, Ampomah Ernest; MARTEY, Ezekiel Mensah; CHRIS, Abilimi Gilbert. **Qualitative assessment of compiled, interpreted and hybrid programming languages**. Communications, v. 7, n. 7, p. 8-13, 2017.

LECUN, Y; CORTES, C.; BURGES, C.J.C. **The MNIST database of Handwritten Digits.** New York, USA. 1998.

MOZILLA. **Javascript, Origem e História.** [S.l.]. 2016?. MDN Web Docs. Disponível em: <https: //developer.mozilla.org/pt-BR/docs/Glossary/JavaScript>. Acesso em: 24 de jan. 2022.

MOZILLA. **Understanding webassembly text format.** [S.l.]. 2017?.MDN Web Docs. Disponível em: <https://developer.mozilla.org/pt-BR/docs/WebAssembly/Understanding_the_text_format>. Acesso em: 30 de jun. 2022.

MOZILLA. **Webassembly format.** [S.l.]. 2017?. MDN Web Docs. Disponível em: <https://research.mozilla.org/webassembly>. Acesso em: 23 de fev. 2022.

MOZILLA. **Performace.now() method.** [S.l]. 2015?. MDN Web Docs. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/Performance/now>. Acesso em: 02 de mai. 2023.

OCARIZA, F. et al. **An Empirical Study of Client-Side JavaScript Bugs.** IEEE International Symposium on Empirical Software Engineering and Measurement, 2013.

OLUWATOSIN, H. S. **Client-Server Model.** IOSR Journal of Computer Engineering, 2014.

OSMANI, Addy. **Javascript Start-up Optimization.** [S.l.]. 2017. Web.dev. Disponível em: <https://web.dev/optimizing-content-efficiency-javascript- startup-optimization/>. Acesso em: 30 de mar. 2022.

OTSU, N. **A Threshold Selection Method from Gray-Level Histograms**. IEEE Transactions on Systems, Man, and Cybernetics, vol. 9, no. 1, pp. 62-66, Jan. 1979, doi: 10.1109/TSMC.1979.4310076.

PARKKINEN, Jaana; HAUKIJÄRVI, Mikko; NENONEN, Petri. **A fast method for scaling color images.** 17th European Signal Processing Conference (EUSIPCO). 2009

RAJU, P. D. R.; NEELIMA, G. **Image segmentation by using histogram thresholding**. International Journal of Computer Science Engineering and Technology, v. 2, n. 1, p. 776-779, 2012.

SARAVANA, C. **Color Image to Grayscale Image Conversion.** 2010 Second International Conference on Computer Engineering and Applications, 2010, pp. 196-199, doi: 10.1109/ICCEA.2010.192.

SURMA, Das. **Is webassembly magic performance pixie dust?.** [S.l.]. 2021. Surma.dev. Disponível em: <https://surma.dev/things/js-to-asc/index.html>. Acesso em: 30 de jun. 2022.

WHATWG. **The canvas element.** [S.I.] 2022. HTML Spec WHATWG. Disponível em: <https://html.spec.whatwg.org/multipage/canvas.html#the-canvas-element>. Acesso em: 24 de ago. de 2022.

WILEY, Victor; LUCAS, Thomas. **Computer vision and image processing: a paper review**. International Journal of Artificial Intelligence Research, 2018.

YOUSEFI, J. **Image binarization using otsu thresholding algorithm**. Ontario, Canada: University of Guelph, 2011.

GLOSSÁRIO

TOOLCHAIN: conjunto de ferramentas usadas no desenvolvimento de software para realizar tarefas complexas.

LLVM: É uma coleção de tecnologias de compilador e toolchains. É usado para otimizar e produzir código intermediário ou código de máquina. A tecnologia também pode ser utilizada como um framework para construção de compiladores.

CLIENT-SIDE: Termo usado no desenvolvimento de sites/aplicativos web para designar o lado do usuário final. Tudo o que o usuário interage.

WSL 2: Maneira oficial de virtualizar um sistema GNU/Linux diretamente no Windows sem modificações, instalação de *dualboot* ou máquinas virtuais tradicionais.

APÊNDICE

APÊNDICE A - Repositório contendo o código do projeto desenvolvido ao longo deste trabalho:

Todo o desenvolvimento do trabalho está disponível no github no seguinte endereço: https://github.com/david123ramos/WASM-Binarization.